<http://databases.about.com/od/specificproducts/a/normalization.htm>

**Normalisation Basics**

If you've been working with databases for a while, chances are you've heard the term normalization. Perhaps someone's asked you "Is that [database](http://databases.about.com/library/glossary/bldef-database.htm) normalized?" or "Is that in [BCNF](http://databases.about.com/library/glossary/bldef-bcnf.htm)?" All too often, the reply is "Uh, yeah." Normalization is often brushed aside as a luxury that only academics have time for. However, knowing the principles of normalization and applying them to your daily database design tasks really isn't all that complicated and it could drastically improve the performance of your DBMS.   
  
In this article, we'll introduce the concept of normalization and take a brief look at the most common normal forms. Future articles will provide in-depth explorations of the normalization process.

**What is Normalization?**

Normalization is the process of efficiently organizing data in a database. There are two goals of the normalization process: eliminating redundant data (for example, storing the same data in more than one [table](http://databases.about.com/library/glossary/bldef-table.htm)) and ensuring data dependencies make sense (only storing related data in a table). Both of these are worthy goals as they reduce the amount of space a database consumes and ensure that data is logically stored.

**The Normal Forms**

The database community has developed a series of guidelines for ensuring that databases are normalized. These are referred to as normal forms and are numbered from one (the lowest form of normalization, referred to as [first normal form](http://databases.about.com/od/specificproducts/l/aa1nf.htm) or 1NF) through five (fifth normal form or 5NF). In practical applications, you'll often see [1NF](http://databases.about.com/od/specificproducts/l/aa1nf.htm), [2NF](http://databases.about.com/od/specificproducts/a/2nf.htm), and [3NF](http://databases.about.com/od/specificproducts/l/aa3nf.htm) along with the occasional 4NF. Fifth normal form is very rarely seen and won't be discussed in this article.   
  
Before we begin our discussion of the normal forms, it's important to point out that they are guidelines and guidelines only. Occasionally, it becomes necessary to stray from them to meet practical business requirements. However, when variations take place, it's extremely important to evaluate any possible ramifications they could have on your system and account for possible inconsistencies. That said, let's explore the normal forms.

**First Normal Form (1NF)**

First normal form (1NF) sets the very basic rules for an organized database:

* Eliminate duplicative [columns](http://databases.about.com/library/glossary/bldef-column.htm) from the same table.
* Create separate tables for each group of related data and identify each [row](http://databases.about.com/library/glossary/bldef-row.htm) with a unique column or set of columns (the [primary key](http://databases.about.com/library/glossary/bldef-primarykey.htm)).

**Second Normal Form (2NF)**

Second normal form (2NF) further addresses the concept of removing duplicative data:

* Meet all the requirements of the first normal form.
* Remove subsets of data that apply to multiple rows of a table and place them in separate tables.
* Create relationships between these new tables and their predecessors through the use of [foreign keys](http://databases.about.com/library/glossary/bldef-foreignkey.htm).

**Third Normal Form (3NF)**

Third normal form (3NF) goes one large step further:

* Meet all the requirements of the second normal form.
* Remove columns that are not dependent upon the primary key.

**Fourth Normal Form (4NF)**

Finally, fourth normal form (4NF) has one additional requirement:

* Meet all the requirements of the third normal form.
* A relation is in 4NF if it has no multi-valued dependencies.

Remember, these normalization guidelines are cumulative. For a database to be in 2NF, it must first fulfill all the criteria of a 1NF database.

**First Normal Form (1NF) sets the very basic rules for an organized database:**

* Eliminate duplicative columns from the same table.
* Create separate tables for each group of related data and identify each row with a unique column (the primary key).

What do these rules mean when contemplating the practical design of a database? It’s actually quite simple.

The first rule dictates that we must not duplicate data within the same row of a table. Within the database community, this concept is referred to as the atomicity of a table. Tables that comply with this rule are said to be atomic. Let’s explore this principle with a classic example – a table within a human resources database that stores the manager-subordinate relationship. For the purposes of our example, we’ll impose the business rule that each manager may have one or more subordinates while each subordinate may have only one manager.

Intuitively, when creating a list or spreadsheet to track this information, we might create a table with the following fields:

* Manager
* Subordinate1
* Subordinate2
* Subordinate3
* Subordinate4

However, recall the first rule imposed by 1NF: eliminate duplicative columns from the same table. Clearly, the Subordinate1-Subordinate4 columns are duplicative. Take a moment and ponder the problems raised by this scenario. If a manager only has one subordinate – the Subordinate2-Subordinate4 columns are simply wasted storage space (a precious database commodity). Furthermore, imagine the case where a manager already has 4 subordinates – what happens if she takes on another employee? The whole table structure would require modification.

At this point, a second bright idea usually occurs to database novices: We don’t want to have more than one column and we want to allow for a flexible amount of data storage. Let’s try something like this:

* Manager
* Subordinates

where the Subordinates field contains multiple entries in the form "Mary, Bill, Joe"

This solution is closer, but it also falls short of the mark. The subordinates column is still duplicative and non-atomic. What happens when we need to add or remove a subordinate? We need to read and write the entire contents of the table. That’s not a big deal in this situation, but what if one manager had one hundred employees? Also, it complicates the process of selecting data from the database in future queries.

Here’s a table that satisfies the first rule of 1NF:

* Manager
* Subordinate

In this case, each subordinate has a single entry, but managers may have multiple entries.

Now, what about the second rule: identify each row with a unique column or set of columns (the primary key)? You might take a look at the table above and suggest the use of the subordinate column as a primary key. In fact, the subordinate column is a good candidate for a primary key due to the fact that our business rules specified that each subordinate may have only one manager. However, the data that we’ve chosen to store in our table makes this a less than ideal solution. What happens if we hire another employee named Jim? How do we store his manager-subordinate relationship in the database?

It’s best to use a truly unique identifier (such as an employee ID) as a primary key. Our final table would look like this:

* Manager ID
* Subordinate ID

**Recall the general requirements of 2NF:**

* Remove subsets of data that apply to multiple rows of a table and place them in separate tables.
* Create relationships between these new tables and their predecessors through the use of foreign keys.

These rules can be summarized in a simple statement: 2NF attempts to reduce the amount of redundant data in a table by extracting it, placing it in new table(s) and creating relationships between those tables.  
  
Let's look at an example. Imagine an online store that maintains customer information in a database. They might have a single table called Customers with the following elements:

* CustNum
* FirstName
* LastName
* Address
* City
* State
* ZIP

A brief look at this table reveals a small amount of redundant data. We're storing the "Sea Cliff, NY 11579" and "Miami, FL 33157" entries twice each. Now, that might not seem like too much added storage in our simple example, but imagine the wasted space if we had thousands of rows in our table. Additionally, if the ZIP code for Sea Cliff were to change, we'd need to make that change in many places throughout the database.  
  
In a 2NF-compliant database structure, this redundant information is extracted and stored in a separate table. Our new table (let's call it ZIPs) might have the following fields:

* ZIP
* City
* State

If we want to be super-efficient, we can even fill this table in advance -- the post office provides a directory of all valid ZIP codes and their city/state relationships. Surely, you've encountered a situation where this type of database was utilized. Someone taking an order might have asked you for your ZIP code first and then knew the city and state you were calling from. This type of arrangement reduces operator error and increases efficiency.  
  
Now that we've removed the duplicative data from the Customers table, we've satisfied the first rule of second normal form. We still need to use a foreign key to tie the two tables together. We'll use the ZIP code (the primary key from the ZIPs table) to create that relationship. Here's our new Customers table:

* CustNum
* FirstName
* LastName
* Address
* ZIP

We've now minimized the amount of redundant information stored within the database and our structure is in second normal form!

There are two basic requirements for a database to be **in third normal form:**

* Already meet the requirements of both [1NF](http://databases.about.com/od/specificproducts/l/aa1nf.htm) and [2NF](http://databases.about.com/od/specificproducts/a/2nf.htm)
* Remove [columns](http://databases.about.com/library/glossary/bldef-column.htm) that are not fully dependent upon the [primary key](http://databases.about.com/library/glossary/bldef-primarykey.htm).

Imagine that we have a table of widget orders that contains the following attributes:

* Order Number
* Customer Number
* Unit Price
* Quantity
* Total

Remember, our first requirement is that the table must satisfy the requirements of 1NF and 2NF. Are there any duplicative columns? No. Do we have a primary key? Yes, the order number. Therefore, we satisfy the requirements of 1NF. Are there any subsets of data that apply to multiple rows? No, so we also satisfy the requirements of 2NF.

Now, are all of the columns fully dependent upon the primary key? The customer number varies with the order number and it doesn't appear to depend upon any of the other fields. What about the unit price? This field could be dependent upon the customer number in a situation where we charged each customer a set price. However, looking at the data above, it appears we sometimes charge the same customer different prices. Therefore, the unit price is fully dependent upon the order number. The quantity of items also varies from order to order, so we're OK there.

What about the total? It looks like we might be in trouble here. The total can be derived by multiplying the unit price by the quantity, therefore it's not fully dependent upon the primary key. We must remove it from the table to comply with the third normal form. Perhaps we use the following attributes:

* Order Number
* Customer Number
* Unit Price
* Quantity

Now our table is in 3NF. But, you might ask, what about the total? This is a derived field and it's best not to store it in the database at all. We can simply compute it "on the fly" when performing database queries. For example, we might have previously used this query to retrieve order numbers and totals:

SELECT OrderNumber, Total

FROM WidgetOrders

We can now use the following query:

SELECT OrderNumber, UnitPrice \* Quantity AS Total

FROM WidgetOrders

to achieve the same results without violating normalization rules.